sevenstax

sevenstax Transmission Control Protocol

TCP Specification and User Manual

Revision No.:	3.11
State:	Released
Author:	Peter Seliger, sevenstax GmbH

Initial version:	21/08/02
Last change:	03/03/03
Changed by:	Ralf Krizsan
Last Review:	13/02/2003
Publication:	Public
Filename:	sevenstaxTCP_UserManual.sxw

Copyright (c) 2003 by sevenstax GmbH
This document is an intellectual property of sevenstax GmbH. Unauthorized copying and distribution is prohibited.

Table of Contents

1 Preface....3
2 Mode of Operation....4
3 Processing Received Packets....6
4 Single and Multiple Connections....7
5 Internal Data Structures....8
6 Public Functions....10
7 Adjustable Parameters....13
8 Integration into software....14
9 Application defined Handler Procedure....18
10 Supported Add-Ons....22
11 Debugging Support....26
12 Ethernet and the Address Resolution Protocol (ARP)....26
13 Restrictions....31
14 Description of Files....32
15 Change History....33

1 Preface

The Internet is today's most disposable information source. On almost every place in the world (if there is a telephone jack), people using computers can access this vast reservoir of information. Normally, these computers are big, fast and powerful. They have large amount of memory, processors faster than light and bloated, full-featured operating systems. Because of those computing power, they easily deal with complex network protocols such as PPP and TCP/IP.

As a contrast to this, connecting smaller devices (such as embedded systems) to the Internet is somehow a tricky and difficult task. The biggest problem is to put such network protocol software into the limited memory space of small microcontrollers. But is there always a need to do so? The answer is: 'No, sometimes it's enough to implement only the basic parts of a protocol'. This is why sevenstaxTCP was made.

For the reason of smallness, the sevenstaxTCP has no additional protocols (by default) which usually belongs to a complete TCP/IP suite of protocols. For example: sevenstaxTCP cannot be 'pinged' because it doesn't know ICMP (but this can be enabled by conditional compilation). Also a sliding window mechanism is not implemented (which is usually part of most TCP stacks) to reduce memory usage. sevenstaxTCP relies on the other end's intelligent control algorithms to improve transmission speed (if it has). Therefore, receiving data with sevenstaxTCP is at an acceptable speed. sevenstaxTCP itself does not have such algorithms nor a retransmission queue thus transmission speed is moderate.

To show for what sevenstaxTCP is good for and when it's better to select another TCP, see the following comparisons:

The advantages of sevenstaxTCP are:

- No receive and transmit buffers. sevenstaxTCP shares buffers with the lower level network protocol (e.g. sevenstaxPPP or an ethernet driver).
- Minimum ROM and RAM usage. Code size differs from 2K to 4K (CISC) depending on the processor where it is executed. This makes it an excellent choice for small embedded systems.
- Pure TCP/IP. No additional protocols such as ICMP, DHCP, UDP etc. are implemented to save memory (But some of them can be enabled by conditional compilation - see the chapter 'Currently Supported Add-ons').
- Easy integration with existing software. Only few interfaces.
- Asynchronous behaviour. Other software will not be delayed by sevenstaxTCP.
- Trouble-free operation under an RTOS environments like CMX and embOS.
- Shipped as well-documented source code (ANSI-C) which makes it easy to modify.

The disadvantages of sevenstaxTCP are:

- Does not follow RFC in all cases.
- No sliding window algorithms. Every single packet is acknowledged to avoid waste of memory and the need of buffer management.

You should give sevenstaxTCP a trial when you...

- ...want to bring the Internet to small embedded devices which suffer from less memory, both RAM & ROM.
- ...want to create HTML browsers or other TCP clients for systems without TCP/IP stack.
- ...want to build devices which are configurable over a TCP/IP network (printers etc.).

2 Mode of Operation

This section describes the way sevenstaxTCP works internally:

- *TCP basics*

  TCP is a connection-oriented protocol. This means that TCP has to compensate these deranging side effects of the underlying network:

  - Packet loss
  - Packet duplication
  - Out-of-order data delivery

  To perform this tasks, some kind of acknowledging and confirmation must be used. For this purpose, TCP packets carry additional information such as sequence/acknowledge numbers and bit flags. Before two TCPs can exchange data, they must synchronize (connect) to each other. This is done by both stations, exchanging a special sequence of datagrams where they tell their initial sequence numbers. During operation, each TCP knows what the other end awaits and tries to deliver the packets that the other TCP has 'ordered'. Also it manages a buffer with already transmitted data to repeat transmission if the recipient hasn't acknowledged the packets. After both TCPs have finished, the connection must be closed. This can be done either by exchanging datagrams similar to the connection process or (faster) by sending a 'connection reset' from any station. Of course, this is a very much incomplete description of TCP/IP (an TCP/IP tutorial is beyond the scope of this document) - but it's nearly all what sevenstaxTCP does.

- *Connection establishment*

  For the purpose of connection establishment, sevenstaxTCP has two interfaces named stxTCP_Connect() and stxTCP_Listen() which take the remote IP address and port number (how a connection is established is shown in RFC 793, figure 7).

  NOTE: For stxTCP_Connect(), the local port numbers of sevenstaxTCP are pseudo-random values.

  sevenstaxTCP tries to send some SYNs to the other end until it receives with a SYNACK. If, for any reason, no SYNACK arrives, sevenstaxTCP notifies the application (through the handler procedure described below) that a connection isn't possible.

  When sevenstaxTCP runs as a server TCP, it is also able to respond to connection attempts issued from remote TCPs. For this to take effect, sevenstaxTCP must be compiled with SERVER_SUPPORTED #defined. Instead of calling

stxTCP_Connect(), the application should call stxTCP_Listen() to bring sevenstaxTCP into the listen state. For further information refer to the chapter 'Currently Supported Add-ons' (Server functionality).

Also sevenstaxTCP has a built-in mechanism to reuse half-open connections. After a possibly system crash, the remote server believes the connection is already open and thus, connecting to the same port will fail. In In this case, sevenstaxTCP sends a RST (TCP reset) to the server, using data the server has sent as an reaction of sevenstaxTCP's connection attempt. The RST cancels the old connection and a subsequent connection request will succeed.

- *Receiving TCP Packets*

  If the lower network layer (e.g. sevenstaxPPP) has recognized a TCP packet, it calls sevenstaxTCP's receive function stxTCP_ProcessPacket(). Before calling this function, the lower network layer prepares a packet descriptor (an internal data structure described below) which holds information about the TCP packet. Usually, the packet resides in the receive buffer of the lower network layer thus sevenstaxTCP doesn't need a buffer itself. The stxTCP_ProcessPacket() function does everything necessary on reception including data delivery to the application, handling connection requests and transmits response packets back to the other end.

- *Transmitting TCP Packets*

  To transmit TCP packets, sevenstaxTCP needs to know two things from the lower network layer (e.g. sevenstaxPPP). First it needs the address of the transmit buffer (a packet descriptor) - sevenstaxTCP must know where to assemble a TCP packet. Next, sevenstaxTCP needs to know the transmit function of the lower layer. This function takes the packet descriptor and forwards the packet to the network. Immediately after sevenstaxTCP has built the packet in memory it prepares the packet descriptor and calls the transmit function to send out the packet.

  Inside sevenstaxTCP there's a function named stxTCP_InternalSend() which does all the above described functions.

- *Terminating a TCP connection*

  Usually, a web server terminates the TCP connection after it has transferred the last byte of the page. When the server wants to close the connection, it sends a FINACK and enters the FIN_WAIT_1 state (see RFC 793, figure 13).

  Users of sevenstaxTCP must call the function stxTCP_Close() which initiates the closing process. When the connection is closed, sevenstaxTCP calls the application's handler procedure by posting a NC_TCP_FIN event.

  There are some other conditions, when sevenstaxTCP issues NC_TCP_FIN as well. For example: If sevenstaxTCP sends data (e.g. an HTTP request) but the packet wasn't acknowledged after a few retries (ACK time-out), the TCP connection will also be terminated. To separate regular terminations from error conditions, the handler procedure is called with different parameters. For a detailed description see the following table (the constants are defined in tcpdefs.h, typedef enum ABORT_REASON):

Cause of termination	*First parameter of handler procedure (ABORT_REASON)*	*Meaning*
A successful call to stxTCP_Close() or the remote TCP has closed the connection.	AR_CLOSED	Normal operation. All data has been successfully transferred.
1. stxTCP_Close() was called and sevenstaxTCP was in transition state (neither in the connected nor in the closed state). 2. The other end has send an RST. 3. Timeout. The remote TCP does not respond to our FINACK's. or FIN's	AR_CLOSE_FORCED	Error. Some data might be received.
Transmitted data was not acknowledged after some retransmissions.	AR_RTM_EXCEEDED	Error. Too many retransmissions.
ARP wasn't able to resolve the other side's MAC address. (Ethernet only)	AR_NO_MACADDRESS	Error. TCP can't continue. Maybe peer does not exist on the LAN.
Application actively terminated the connection.	AR_USER_1 AR_USER_2 AR_USER_3	Normal operation. The TCP connection was cancelled.

3 Processing Received Packets

The nearly most important attribute of a TCP is how it handles received packets. In fact, sevenstaxTCP handles received packet in a different way than most other TCPs. The sequence in which packets are processed is shown here:

1. First, protocol and port number are validated. If they don't match, the packet is rejected. If the protocol is UDP and UDP_SUPPORTED is #defined, the packet is forwarded to stxTCP_ProcessUDP(). Otherwise, if the protocol is ICMP and ICMP_SUPPORTED is #defined, the packet goes to stxTCP_ProcessICMP().
2. If we #defined SERVER_SUPPORTED and the flags field of the TCP header contains a SYN only, sevenstaxTCP sends back a SYNACK and enters the internal SYN_RECEIVED state. After that, further packet processing is not necessary at this point.
3. **Please Note:** sevenstaxTCP's SYN_REVEIVED state has a slightly different meaning than those described in RFC 793.
4. Sequence- and acknowledge number from the packet are stored in global variables.
5. If there is a pending transmission, the received sequence number will be compared with the one expected in order to cancel retransmissions and to free the retransmission buffer so as new data can be send.

6. If a connection is established, a time-out value to cancel a dead connection (a connection where no data has been received for a long time) will be checked first.
7. If a connection is established, the received acknowledge number will be compared with the calculated one. This means we have received a packet in the right order which will then be delivered to the upper layer (if it carries data). Also the acknowledge value for the next packet will be calculated here.
8. If a connection is established, and the received acknowledge number does not match the expected one, the global acknowledge number will be set to the one awaited and an ACK segment with this number is send to the server to force a retransmission. At this point, the handling of this packet will be aborted.
9. If sevenstaxTCP receives a SYNACK (after a SYN has been sent), an ACK will be returned, the alive timer initialized, the first acknowledge number calculated and the application notified that we the connection is established and data may be transferred. At this point no further packet will be handled.
10. If sevenstaxTCP receives a FINACK (e.g. the server wants to close the connection) an ACK is returned and the connection aborted. Furthermore the application will be notified of the connection's termination. No further packet handling will be executed.
11. If sevenstaxTCP receives a RST (e.g. the server has detected a TCP error) the connection will be aborted and the application notified of this event. No further packet handling will be done.
12. In case an ACK is received while a connection is established, a data packet must be acknowledged. sevenstaxTCP does this and leaves the receive function.
13. In case an ACK is received while no connection is established this may indicate that a half-open connection exists. sevenstaxTCP will send an RST which will close the old connection enabling a new one to be established on the same port.

4 Single and Multiple Connections

By default, sevenstaxTCP runs in single connection mode. This is useful for most embedded devices, where memory is a very limited resource. In cases where more than one simultaneous connections must exist (e.g. FTP), sevenstaxTCP can be configured to work with up to 65535 connections at a time. Multi connection support can be switched on at compile time by #defining TCP_MULTI and TCP_MULTI_TICKS.

With TCP_MULTI one can specify the maximum number of connections which sevenstaxTCP can handle. This should be well adjusted to save memory because for every connection about 70 bytes are reserved.

TCP_MULTI_TICKS is the number of connections which are handled by a single call to stxTCP_Tick(). This value also depends a little how frequent stxTCP_Tick() is called. For example: If you configure sevenstaxTCP to work with 100 possible connections, TCP_MULTI_TICKS can be set to 10.

Both #defines can be found in 'features.h'. Please delete or comment them out (/**/), if you only need one connection.

Most of sevenstaxTCP's interfaces has an additional parameter called 'Connection-ID'. Connection ID's are unique values which help to distinguish between connections. They are unique during the lifetime of a connection. Connection IDs are automatically recycled (newly assigned) when a connection ends.

Please Note: TCP_MULTI_TICKS does not affect incoming TCP data. Received packets are handled immediately.

5 Internal Data Structures

- *The packet descriptor*

```
typedef struct _tag_PACKET_DESC
{
     PACKET_STATUS state;
     UINT16 size;
     UINT8 *data;
     UINT8 *payload;
} PACKET_DESC;
```

The packet descriptor is used to hold information about a data packet and for easy passing of this information between internal functions by reference. The packet status member (PACKET_STATUS state) indicates whether the packet is in use (i.e. under construction) or ready for further processing. It is for internal use only, must not be initialized or changed by the application or lower network components. The member '*data' points to the location of the first byte in memory. This is the place where the packet's data is stored including link layer headers. The member 'size' specifies the length in bytes of the entire range pointed to by '*data'. Finally, the member '*payload' points to the first byte of the IP header. This is usually the first byte after the end of the link layer header.

Packet descriptors are also used by sevenstaxPPP to deliver packets to higher network protocols (such as TCP).

- *The IP address*

```
typedef union _tag_IPV4
{
     struct
     {
            UINT8 b1;
            UINT8 b2;
            UINT8 b3;
            UINT8 b4;
     } ipb;
     UINT32 ipl;
} IPV4;
```

An IP address is actually a 32 bit value which is often shown as 4 decimal values, separated by dots, for better readability (e.g. 192.168.0.51). IP addresses are stored in network byte order where the first byte (ipb.b1) is the most significant byte. The IPV4 union allows either access to the entire value (ipl) or partly through ipb.b1 to ipb.b4.

Please Note: IP addresses are internally stored as they came over the network (network byte order), regardless of the processor architecture sevenstaxTCP runs on.

- *The TCP information structure*

```
typedef struct _tag_TCP_INFORMATION
{
    UINT8 *data;
    UINT16 datasize;
    UINT16 remote_port;
    UINT16 local_port;
    IPV4 remote_ip_address;
} TCP_INFORMATION;
```

This structure is mainly used to pass all important properties regarding to a TCP connection. E.g. when sevenstaxTCP has posted an NC_TCP_CONNECTED event to the application's handler routine, some members (remote_ip_address, remote_port and local_port) contain values which describe the connection. If data arrives on a specific port, sevenstaxTCP issues a NC_TCP_RECV event with an additional pointer to a TCP_INFORMATION structure where (beside the other members) *data and datasize describe a newly received packet.

- *The MAC address*

```
typedef union _tag_MAC_ADDRESS
{
    struct
    {
           UINT8 mb1;
           UINT8 mb2;
           UINT8 mb3;
           UINT8 mb4;
           UINT8 mb5;
           UINT8 mb6;
    } macbytes;          // <---- Use this for single byte access
    struct
    {
           UINT16 mw1;
           UINT16 mw2;
           UINT16 mw3;
    } macwords;          // <---- Use this for symmetric access
    struct
    {
           UINT32 ml;
           UINT16 mw;
    } macx;                     // <---- Use this for very fast access on 32 bit mcus
} MAC_ADDRESS;
```

A media access control (MAC) address is used as an unique identifier of every network interface card (NIC). MAC addresses are used by the Ethernet and ARP modules. There are three different access methods (through various structs of the union).

Please Note: As there is no need for calculations with MAC addresses they are internally stored as received over the network (i.e. in network byte order), regardless of the processor architecture sevenstaxTCP runs on.

6 Public Functions

sevenstaxTCP has a few public interfaces. Some of them are only available if additional features are enabled (such as UDP or server functionality) Please see the following descriptions for details:

- *void stxTCP_Tick (void)*

  This function keeps sevenstaxTCP alive. stxTCP_Tick() must be called as often as possible in order to provide sevenstaxTCP with processor time. It is designed to be very fast thus applications are not slowed down. The most time-consuming action stxTCP_Tick() might do is to build an entire TCP packet (only one per call) and deliver it to the link layer. Therefore, the maximum time stxTCP_Tick() will use depends on the size of the biggest packet (defined by RTM_MAXDATA, see the next chapter).

- *UINT16 stxTCP_Connect (IPV4 *ip_address, UINT16 port)*

  TCP_Connect() tries to establish a connection to another TCP. The first parameter (*ip_address) is a pointer to an IPV4 structure which must be initialized to the IP address of the remote peer. The second parameter (port) is the port number of the remote TCP on which the connection should be established. Ports are something like channels dedicated to particular services. E.g. HTTP resides on port 80 (decimal). Upon a call to stxTCP_Connect(), successful or not, the application-defined handler procedure is called. stxTCP_Connect() is usually called when sevenstaxTCP acts as a client TCP. Server TCPs should call stxTCP_Listen() instead.

  On function return, the caller receives a connection-ID. Subsequent calls into sevenstaxTCP must use this value to identify the connection. Also events, which the application receives by its handler procedure carry connection IDs.

- *void stxTCP_Init (PROTOCOL_NOTIFY_HANDLER handler)*

  Before sevenstaxTCP can be used, this function must be called. It initializes all internal states of sevenstaxTCP and also registers the application-defined handler procedure.

  Please Note: It is advisable to initialize the link layer and set up the receive and transmit packet descriptors before calling stxTCP_Init(). Please see section 7 for further information.

- *BOOL stxTCP_Write (UINT8 *data, UINT16 len, UINT16 connection_id)*

  When a connection is established, stxTCP_Write() can be used to send data to the remote station.

  The first parameter points to the first byte of the data that should be send, the second one is the size of the data in bytes. The third parameter is a connection-ID which can be obtained either by a call to stxTCP_Connect() or stxTCP_Listen().

  TCP_Write() only succeed if a connection is established, the data is not too big (see RTM_MAXDATA) and no transmission is currently in progress. If stxTCP_Write() has

accepted a request to send, it returns TRUE, otherwise FALSE. Data is send the next time stxTCP_Tick() is called.

Note: Do not call this function when server functionality is enabled. It will work but you don't get any notification if the data has been successfully transmitted or not. Use stxTCP_WriteBulk() instead.

The data (and possibly a reference) must remain constant until stxTCP_Write() has finished (which is signaled by a NC_TCP_TXREADY event, see below). Because sevenstaxTCP is designed to run on small embedded systems, it doesn't copy the data but operates on the source to avoid wasting memory.

- *BOOL stxTCP_Close (UINT16 connection_id)*

  This function can be called by the application to end a TCP connection. sevenstaxTCP does all necessary actions to close a TCP connection gracefully. The application receives the result as a call to its handler procedure (NC_TCP_FIN).

  A return value (TRUE or FALSE) tells the caller whether closing can be performed or not. Usually stxTCP_Close() only fails when the connection is already in its closed state.

- *UINT16 stxTCP_Listen (UINT16 port)*

  TCP_Listen() is only available if server functionality is enabled. A call to stxTCP_Listen() puts sevenstaxTCP into the listen state, which means that it waits for a connection attempt on the specified port. If another TCP sends a SYN, sevenstaxTCP handles it properly and enters the connected state. stxTCP_Listen() only succeed if sevenstaxTCP is in the closed state, otherwise stxTCP_Listen() returns FALSE.

  The return value is a connection ID which identifies a connection switched to its listen state. It must be used in subsequent calls where connection-ID's are needed. Please see stxTCP_Connect() also.

- *BOOL stxTCP_WriteBulk (UINT8 *data, UINT32 len, UINT16 connection_id)*

  TCP_WriteBulk() is only available if server functionality is enabled. This function is similar to stxTCP_Write() except that it is able to send huge amounts of data to the remote station. Data is fragmented into RTM_MAXDATA-sized pieces and sent out using multiple calls to stxTCP_Write(). Another difference to stxTCP_Write() is, that NC_TCP_TXREADY is issued only after the last fragment was successfully transmitted (and acknowledged). stxTCP_WriteBulk() will fail (return FALSE) if a bulk write is already in progress.

- *void stxTCP_ListenUDP (UINT16 port, BOOL active)*

  TCP_ListenUDP() is only available if UDP_SUPPORTED is switched on (see section 9, currently supported add-ons). Calling this function will enable sevenstaxTCP to receive UDP packets on the specified port. The second parameter is used to switch the receipt on or off. If this parameter is 'TRUE' the listen mode will be enabled and will be disabled if the parameter is 'FALSE'.

Please note that it is possible to use TCP and UDP at the same time both either listening or

- *BOOL stxTCP_WriteUDP (IPV4 *ip, UINT16 port, UINT8 *dataptr, UINT16 datalen)*

  TCP_WriteUDP() is also only available if UDP_SUPPORTED is switched on. You must supply the target's IP address, its port number, a pointer to the data and the size of data (in bytes). stxTCP_WriteUDP() only works after a previous call to stxTCP_ListenUDP() because sevenstaxTCP obtains its local UDP port number as a parameter of this function.

  Please Note: Currently there is no sanity check of the parameters you supply to stxTCP_WriteUDP(). So be careful.

- *BOOL stxTCP_QueryDNS (IPV4 *dns_ip, STRING hostname)*

  If you have DNS enabled (please see section 9, currently supported add-ons), this function instructs sevenstaxTCP to transmit DNS queries on UDP port 53 to the given IP address. The IP address must belong to a DNS machine and the hostname must be a valid name, for example 'www.yahoo.com'. stxTCP_QueryDNS() returns immediately. The result is posted as a notification code to the handler procedure of sevenstaxTCP, which can either be NC_DNS_IP (success) or NC_DNS_TIMEOUT (failure). The return value of stxTCP_QueryDNS() is TRUE, when the query packet was successfully delivered to UDP. It is FALSE when a DNS query is already in progress - this means that neither the DNS answered nor a time-out occurred.

- *void stxNTP_Query (IPV4 *ntp_server);*

  If the NTP (Network Time Protocol) is used by #defining NTP_SUPPORTED in 'features.h'. NTP queries will be started by calling this function. The parameter, stxNTP_Query() takes, is a pointer to an IPV4 structure representing the IP address of the time server. stxNTP_Query() sets some internal variables and returns immediately. The actual query will be done the next time stxTCP_Tick() is called. After a call to stxNTP_Query(), the application defined handler procedure will either receive an NC_NTP_SUCCESS (got the time) or NC_NTP_FAILURE (no response from the time server). If NTP_SUPPORTED is not #defined, this function does not exist to save program memory space.

  Please note: Calls made to stxNTP_Query() while being in progress will re-trigger the query process. Please see chapter 'Currently Supported Add-ons' for a more detailed description of the NTP.

- *void stxNTP_Transform (UINT32 ntps, UINT8 *sec, UINT8 *min, UINT8 *hour, UINT16 *year, UINT8 *month, UINT8 *day, UINT8 *day_of_week);*

  Because the result of an NTP query is a 32bit value of seconds since 1.1.1900 this function is for convenient conversion of this infamous seconds into time and date values. stxNTP_Transform() takes the NTP seconds (ntps) and writes the results into the locations, where the pointers *sec, *min, *hour, *year, *month, *day, and *day_of_week point to. As stxNTP_Query(), this function is also only available if NTP_SUPPORTED is #defined in 'features.h'.
  The time returned by the servers is UTC () which means it has to be corrected if the local time zone differs from UTC. This correction must be done before calling stxNTP_Transform(). The difference between the local time and UTC has to be added

to the seconds returned by the server. After calling NTP-Transform() the right values can be read from the parameters as described above.

7 Adjustable Parameters

There are some parameters defined in 'tcpdefs.h' which can be customized. These parameters should properly be adjusted depending on the lower network layer and communication device. The following list describes them in detail:

- *RTM_TIMEOUT* *default: 6000*

  Retransmission time-out. This is the time in milliseconds within a transmitted packet must be acknowledged. If sevenstaxTCP does not receive an ACK, the packet will be resend.

- *RTM_MAXDATA* *default: 512*

  The size of a single packet which will be accepted by stxTCP_Write() (sevenstaxTCP's send function).

- *RTM_MAXRETRIES* *default: 4*

  This is the number of attempts sevenstaxTCP will make to transmit a single packet to the other end.

- *TCP_PORT_START* *default: 6381*

  Every TCP must be aware of multiple channels, the so-called 'ports'. The local and the remote port number are essential information for TCP connections. With TCP_PORT_START one can specify the first port number sevenstaxTCP uses for its own. (**NOTE:** The remote port number is specified in stxTCP_Connect). Port numbers are 16 bit values therefore they can vary from 0 to 65535. Because port numbers below 1024 are assigned to well known services (see RFC 1340), it is not advisable to use them.

- *TCP_PORT_END* *default: 6391*

  This is the last port number, sevenstaxTCP will use as its local port. For a detailed description see 'Cyclic Port Changing' of chapter 'How it works'.

- *TCP_WINDOW_SIZE* *default: 512*

  TCP_WINDOWSIZE stands for the maximum number of bytes a TCP can receive. This value will be directly written to the TCP header. Normally, a TCP sets this value dynamically according to its free buffer space. Because sevenstaxTCP does not have buffer management, TCP_WINDOW_SIZE is a fixed value.

- *TCP_MULTI* *default: <none>*

If multiple TCP connections are required this define specifies the maximum number of simultaneous connections. For a single connection TCP (by default), TCP_MULTI is commented out.

TCP_MULTI is #defined in 'features.h'.

- *TCP_MULTI_TICKS* *default: <none>*

  If multiple connections are needed, this specifies the number of connections which are handle by a single call to stxTCP_Tick(). For a single connection TCP (by default), TCP_MULTI_TICKS is commented out.

  TCP_MULTI_TICKS is also #defined in 'features.h'.

- *UDP_MULTI* *default: <none>*

  If multiple UDP connections are required this define specifies the maximum number of simultaneous connections. For a single connection UDP (by default), UDP_MULTI is commented out.

  TCP_MULTI is #defined in 'features.h'.

8 Integration into software

sevenstaxTCP is designed to work in conjunction with different data link layers such as PPP or Ethernet. There are some function prototypes and definitions in 'features.h' which must be adjusted.

Please note: Various functions deal with packet descriptors (see 'Internal Data Structures'). The link layer has to provide the required packet descriptors.

- *Access to system-supplied timer/counter* *GET_COUNTER_VALUE*

  To give an example: If there is a function named GetTickCount() to access an internal timer the adjusted line in 'features.h' will look like this:
  #define *GET_COUNTER_VALUE* *GetTickCount()*

  For some purposes, such as calculating time-outs etc. sevenstaxTCP needs to access a free running 32 bit timer/counter with a resolution between one millisecond to one second. The entry GET_COUNTER_VALUE in 'features.h' should reference a function or a register (e.g. of a timer chip or microcontroller) where this value can be fetched. The default entry currently references the function GetTickCount() from MS-Windows. This must be changed to satisfy your needs.

- *C-library strlen() function* *GET_STRING_LENGTH*

  Example: #define *GET_STRING_LENGTH* *strlen()*

  sevenstaxTCP needs a function to get the length of a zero-terminated byte array (c-string). If your environment supports the c-library strlen() function then you can leave GET_STRING_LENGTH unchanged, otherwise you must write a similar function yourself and redefine GET_STRING_LENGTH to reflect your change.

- *Data reception TCP_RECV*

  *Example: #define TCP_RECV stxTCP_ProcessPacket (PACKET *pkt)*

Whenever the link layer (e.g. PPP or Ethernet) has detected a TCP packet, it must be delivered to sevenstaxTCP. Therefore, the link layer has to know the entry point into sevenstaxTCP where received packets are handled. For this reason sevenstaxTCPs internal function stxTCP_ProcessPacket() is made public by the macro TCP_RECV. The supplied packet descriptor must have the following settings when entering TCP_RECV:

- state: Not used (any value).
- size: Size of the entire packet (including link layer headers).
- data: A pointer to the first byte of the link layer header.
- payload: A pointer to the first byte of the IP header.

Note: The layout of a packet descriptor is shown in section four.

- *Data transmission SEND_PKT*

  *Example: #define SEND_PKT LinkLayer_SendFrame (PACKET *pkt)*

This is the entry point of the link layer, which is responsible for handling data that should be sent off to the network. sevenstaxTCP requires a function which takes a pointer to a packet descriptor as a single parameter. When sevenstaxTCP calls SEND_PKT it has done these things before:

- It has build a complete TCP/IP packet beginning at the location where the packet descriptor's member '*payload' points to.
- It has set the packet descriptor's 'size' member to the size of the TCP/IP packet.

The link layer might then do the following:

- Increase the 'size' member to the length of its own headers.
- Build its headers at the location where '*data' points.
- Deliver the packet to the network hardware.

- *Packet copy function COPY_PKT*

  *Example: #define COPY_PKT CopyPkt (PACKET *src, PACKET *dest)*

CopyPkt() resides in the file 'runtimelib.c'. The function behind COPY_PKT copies the 'real' information content from a source packet descriptor (which means size, state and the entire data) to a destination packet descriptor.

Warning: Source and destination packet descriptors MUST have valid data pointers and a valid size information otherwise data is written elsewhere and your system will probably malfunction.

- *Send buffer SENDBUFF*

  Example: #define SENDBUFF gPPP_sendpacket

sevenstaxTCP is designed to reuse the send buffer of the link layer (e.g. sevenstaxPPP) in order to save memory. Therefore SENDBUFF is by default set to use sevenstaxPPP. This behaviour is changed implicitly by defining RUNS_ON_ETHERNET which is also defined in 'features.h'. If sevenstaxTCP is used to transfer data via an Ethernet network the line above is automatically to

#define SENDBUFF gETH_sendpacket.

SENDBUFF is a packet descriptor structure (see section 4) and must be initialised the following way:

- The '*data' member must point to first byte of the link layer header. This might be the first byte of the buffer which the link layer will send off to the network.
- The '*payload' member must point to the first byte of the IP header. This should be, for example, fourteen bytes behind '*data' when using sevenstaxTCP on Ethernet.
- 'size' and 'state' members are not used at this point and can have any value.

The buffer can be changed according to the projects needs. Therefor the SENDBUFF may be changed to a different packet descriptor but this is normally not required.

- *Local IP address LOCAL_IP_ADDRESS*

  Example: #define LOCAL_IP_ADDRESS gTCP_my_ipaddress

As every TCP sevenstaxTCP needs an IP address of its own. The local IP address is declared in sevenstaxTCP and made public trough LOCAL_IP_ADDRESS. When sevenstaxTCP operates with sevenstaxPPP, The IP address will dynamically assigned by the Internet service provider. This means that you must not change it. Otherwise, when sevenstaxTCP is used in another environment (e.g. with Ethernet), this variable (an IPV4 structure) must be initialized either manually or by an network control protocol such as DHCP.

Please Note: LOCAL_IP_ADDRESS is not initialized by stxTCP_Init() and may contain random values.

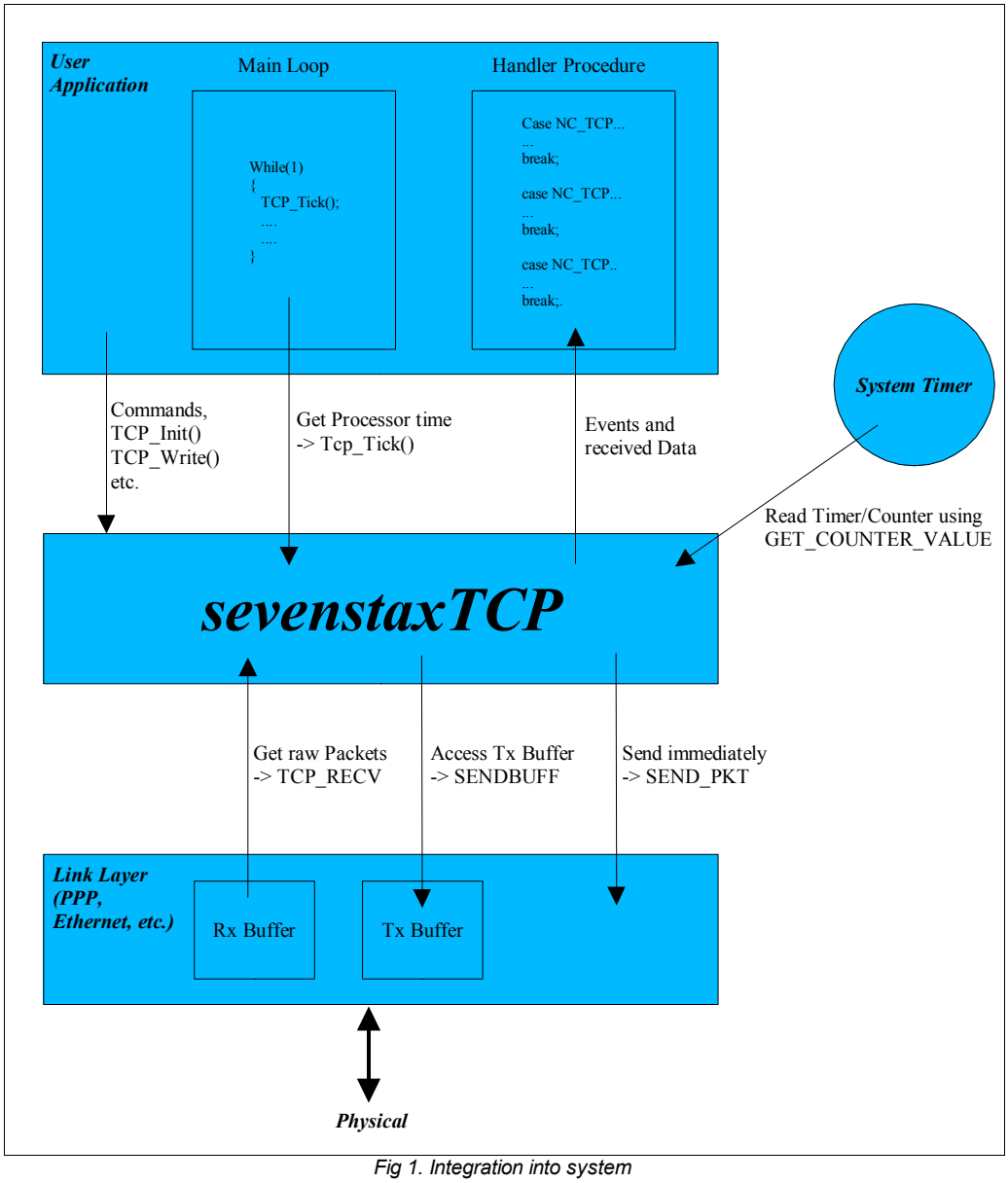

Fig 1. Integration into system

9 Application defined Handler Procedure

To signal events, sevenstaxTCP needs a handler procedure provided by the application. Whenever sevenstaxTCP has something to report, it calls this handler procedure. By calling stxTCP_Init(), the application registers such a handler procedure with sevenstaxTCP. Because sevenstaxPPP also needs a similar handler procedure, the definition of it will be found in 'notifycodes.h' and looks like this:

```
typedef void (*PROTOCOL_NOTIFY_HANDLER)(NOTIFY_CODE code, UINT32 lparam, UINT16 wparam)
```

The first parameter is called the NOTIFY_CODE. This is actually an identifier of the message which sevenstaxTCP has sent. The next two parameters may carry additional information depending on the message. NOTIFY_CODE is an enumeration type and is also defined in 'notifycodes.h':

```
typedef enum
{
     NC_PPP_INITIALIZED = 1,              // PPP ...
     NC_PPP_CONNECTED,                    // PPP ...
     NC_PPP_CONNTIMEOUT,                  // PPP ...
     NC_PPP_SERVER_TERMINATED,            // PPP ...
     NC_PPP_PRIMARY_DNS,                  // PPP ...
     NC_PPP_SECONDARY_DNS,                // PPP
     NC_TCP_CONNECTED,                    // TCP connection established
     NC_TCP_CONNTIMEOUT,                  // TCP connection timed out
     NC_TCP_FIN,                          // TCP remote peer disconnected
     NC_TCP_RECV,                         // TCP data available
     NC_TCP_TXREADY,                      // TCP ready to send
     NC_UDP_RECV,                         // UDP data received
     NC_ARP_HOSTMESSAGE,                  // Got ARP response or query
     NC_ARP_TIMEOUT,                      // No ARP response received
     NC_DNS_IP,                           // Got DNS response
     NC_DNS_TIMEOUT,                      // No DNS response received
     NC_NTP_SUCCESS,                      // Got time server response
     NC_NTP_FAILURE,                      // NTP failed for any reason
} NOTIFY_CODE;
```

With one of the NC_TCP_XXX codes, sevenstaxTCP will call the handler procedure to inform about occurred events. Below is a short description of their meanings. By the way, the first four codes of the enum refer to sevenstaxPPP and are not used by sevenstaxTCP and thus are not described here.
Note: Notify codes of both modules are combined so that a single handler procedure may be shared by sevenstaxTCP and sevenstaxPPP.

- *NC_TCP_CONNECTED*

  After a successful call to stxTCP_Connect(), NC_TCP_CONNECTED is issued. The application can now send and receive data to/from sevenstaxTCP. Both parameters contain port numbers copied directly from the received packet. The first parameter (lparam) points to a TCP_INFORMATION structure which describes the newly created connection. The members 'remote_ip_address', 'remote_port' and 'local_port' are valid.

  The second parameter (wparam) contains a connection-ID..

- *NC_TCP_CONNTIMEOUT*

  This event occurs if stxTCP_Connect() failed (i.e. timed out). sevenstaxTCP does a few retries to establish a connection and then gives up.

  The first parameter (lparam) is always zero (not used) and the second parameter is a connection-ID which identifies the connection that failed.

- *NC_TCP_FIN*

  This event is posted when the TCP connection is terminated. This may happen when a web server has finished sending data, when an error occurs, or when the application has called stxTCP_Close().

The first parameter (lparam) is the ABORT_REASON (see chapter 2: Terminating a TCP connection). The second parameter is a connection-ID which identifies the connection that has been closed.

- *NC_TCP_RECV*

  When sevenstaxTCP has received a data packet, TCP_RECV is issued to tell the application that data has arrived.

  The first additional parameter points to a TCP_INFORMATION structure which describes the incoming data packet. All members are valid. The second parameter is a connection-ID which identifies the connection.

- *NC_TCP_TXREADY*

  This event is posted to signal the application that sevenstaxTCP is ready to accept data for transmission. It occurs after sevenstaxTCP has received an ACK as an answer to transmitted data. Both parameters are not used. After getting NC_TCP_TXREADY, the buffer allocated for stxTCP_Write() or stxTCP_WriteBulk() can be reused/freed. There is a little difference when sevenstaxTCP runs in server mode (SERVER_SUPPORTED in 'features.h') - NC_TCP_TXREADY is then issued after sevenstaxTCP has transmitted the entire range of data which was specified through a call to stxTCP_WriteBulk(). Otherwise if sevenstaxTCP is in client mode every successful transmission of a single packet will cause a NC_TCP_TXREADY.

  The first parameter (lparam) is not used and always zero. The second parameter (wparam) contains a connection-ID.

- *NC_UDP_RECV*

  When sevenstaxTCP's optional UDP support is enabled, this event is generated as a result of a successfully received UDP packet from the remote machine. Just as with TCP data, the first additional parameter points to the first byte and the second parameter indicates the size (in bytes) of the packet.

- *NC_ARP_HOSTMESSAGE*

  When sevenstaxTCP runs on Ethernet, the ARP module causes this event to notify the application of a received ARP packet. The first parameter indicates the type of ARP message received. This parameter is either ARP_OP_QUERY when a remote computer has send a response to an ARP query which sevenstaxTCP has originated. or ARP_OP_REPLY when sevenstaxTCP has send an ARP response to another computer as a result of receiving a query. The second parameter is not used and always zero.

- *NC_ARP_TIMEOUT*

  When sevenstaxTCP runs on Ethernet, The ARP module causes this event after the application has called stxARP_Query() and there was no reply within ARP_QUERY_TIMEOUT milliseconds. The time-out value is defined in 'arp.c' and is currently set to 0.5 seconds. If sevenstaxTCP runs on Ethernet as a client, usually stxARP_Query() must be called before actively connecting to another TCP (This means before calling stxTCP_Connect()).

- *NC_DNS_IP*

  After a successful call to stxTCP_QueryDNS(), this event was triggered when the DNS resolver has received an answer and found a record which contains the requested IP address. The IP address can then be read from the first parameter of the handler procedure. The address is passed in network byte order. The second parameter is not used.

- *NC_DNS_TIMEOUT*

  This event is posted as a negative result when stxTCP_QueryDNS() was called and the DNS resolver timed out. There are various reasons for such a failure. The most likely reasons are, that the DNS server is not able to find the requested hostname (e.g. because it does not exist anymore) or the DNS server itself is temporarily not available.

- *NC_NTP_SUCCESS*

  When using NTP, this event indicates an successful reception of a time server response. The first parameter contains the time in seconds since 01.01.1900. The second parameter is not used and may contain random values. NC_NTP_SUCCESS can only occur if NTP_SUPPORTED is #defined in 'features.h'.

- *NC_NTP_FAILURE*

  This is the opposite to NC_NTP_SUCCESS. When an NTP query was made, NC_NTP_FAILURE is posted to the handler procedure if the time server does not respond properly. Both additional parameters have no meanings in this case.

There are three general rules for executing code inside the handler procedure:

- **Leave as fast as possible**

  When sevenstaxTCP signals events, it cannot proceed before the handler procedure has exited. If you do lengthy tasks inside the handler procedure, the receive buffers may overrun, packets get lost and must be repeated. This will cause appreciable delays and might affect performance of the whole system. So be sure not to waste much time here.

- **Never call sevenstaxTCP directly**

  Because the handler procedure is called from inside sevenstaxTCP, it is dangerous calling back into sevenstaxTCP. Doing so can cause deadlocks or even unlimited recursions which will result in stack overflows and thus crash the system.

- **Do not write to RX/TX buffers**

sevenstaxTCP might be in a state where it waits for some data or builds a packet for transmission. Therefore, changing buffer contents can cause sevenstaxTCP to misbehave.

10 Supported Add-Ons

sevenstaxTCP can be extended through various components. This section will grow if new features are added.

- *Ethernet support and Address Resolution Protocol (ARP)*

  Originally, sevenstaxTCP was designed to download data from web servers using a dial-in connection. As some of our customers needed a tiny TCP implementation which uses Ethernet as the link layer, we made it that flexible and integrated Ethernet and ARP support.

- *ICMP echoes*

  Sometimes it may be useful that a TCP responds to ICMP echo requests (also known as 'PING'). sevenstaxTCP can be extended to reply to such a request by #defining ICMP_SUPPORTED. No additional source files are needed. The ICMP support of sevenstaxTCP is very simple and small in code size therefore enabling it will not waste much memory.

  If an incoming packets is identified as an ICMP echo request, sevenstaxTCP does the following steps:

  - The packet is copied to the TX buffer and the type is changed to an ICMP echo reply.
  - A new checksum is calculated.
  - Source and Destination addresses are swapped.
  - The packet is send off to the network.

- *UDP*

  UDP is the abbreviation for User Datagram Protocol, an unreliable packet delivery service, which is part of the TCP/IP suite of protocols. If you need UDP please do the following:

  - In 'features.h' uncomment (or insert): #define UDP_SUPPORTED 1 and the TCP/IP stack has to be rebuild.
  - To receive UDP packets stxTCP_ListenUDP (port number) must be called.
  - To send UDP packets stxTCP_WriteUDP() supplied with the IP address and port number of the remote TCP/IP stack has to be called. parameters of this function is a pointer to the data and the size of data to transmit.

Please note: stxTCP_WriteUDP() can only be used after a single call to stxTCP_ListenUDP() because it needs a local port number.

- Received UDP packets are routed to the application defined handler procedure. The belonging NOTIFY_CODE is NC_UDP_RECV (see chapter 'Application defined Handler Procedure').

There is a little restriction on UDP packets. As fragmentation is not supported. sevenstaxTCP silently throws away all packets that are fragmented - but that is no real drawback because sevenstaxTCP receives not fragmented packets up to the maximum length of the link layer (PPP, Ethernet etc.). Nevertheless care must be taken before building the TCP/IP stack. If the size of the IP packets (define TCP_WINDOW_SIZE in tcpdefs.h) is chosen larger than the size the link layer is able to transmit without fragmentation it might be possible that no data will be received while the sending of data works without problems.

- *Delayed Acknowledgement*

  Roughly all TCP implementations, not only those tiny ones like sevenstaxTCP, often send packets which are pure ACKs. An ACK is a single bit flag but it must be embedded into a full TCP/IP packet to be accepted by the receiver. Also the link layer adds some overhead for the same reason. Regarding only to TCP/IP, there are 40 bytes which must always be transmitted over the network in order to send a single ACK.

  To reduce the amount of such ACK-only messages, one can retard those ACK's and append user data to the packet. This technique can save up to 20% of bandwidth for the price of a slightly reduced speed.

  To enable the benefits of delayed acknowledgment, two things must be done:

  1. #define DELAYED_ACK_SUPPORTED in 'features.h'.
  2. Set an appropriate delay value (ACKDELAY_TIME in 'tcpdefs.h').

  The value of ACKDELAY_TIME (in milliseconds) rather depends on the speed of the underlying network. Using an 19200 bps modem, 10...20ms are a good choice. If you are running sevenstaxTCP on an Ethernet network, ACKFDELAY_TIME should be smaller. As this value depends on the hardware of the system it is difficult to give recommendations. It is a good way to start using the value mentioned above and to decrease it until the best performance of the system is achieved.

- *Server functionality*

  sevenstaxTCP is capable to act as a server TCP. This means it can passively wait for connections from other TCP's and send requested data or error messages to them. To enable the server functionality the following steps are required:

  - In 'features.h' uncomment (or insert): #define SERVER_SUPPORTED 1 and recompile.

- After initialization of sevenstaxTCP call stxTCP_Listen(). This causes sevenstaxTCP to listen on a specified port for incoming connections.
- After being connected, which will be indicated by the received NOTIFY_CODE NC_TCP_CONNECTED call stxTCP_WriteBulk() to send data of nearly any size to the remote TCP.
- When the entire data block was successfully transmitted, more data may be sent data or the connection may be closed by calling stxTCP_ServerAbort() for a graceful - or stxTCP_Abort() for a fast disconnect.
  Please Note: stxTCP_ServerAbort() must be used when a web browser loads a page (i.e. sevenstaxTCP is used by a web server) or otherwise the user might see an error message.

- *Integral Domain Name Service Resolver (DNS)*

  For humans it is rather convenient to work with readable names than with IP addresses. For example the URL 'www.yahoo.com' to access a website instead of of using the IP addresses 64.58.79.230 is much more memorable. That is the reason why DNS was created. A DNS or domain name service, converts names to their corresponding IP addresses.

  The modern DNS is organised a bit like a tree. There is a root called '.' with certain main branches called 'top level domains' (TLD). At the root are a few servers (e.g. a.root-servers.net, b.root-servers.net ...) which know all addresses of all the servers which handle individual TLDs (Top Level Domains) such as .com, .edu, .mil or .net.

  Communication with the DNS is done by UDP on port 53. DNS client software (so called resolvers) must send special packets (see RFC 1035) to the DNS. The DNS, on his part, sends back a packet containing some records which contain e.g. the IP address.

  To use sevenstaxTCP's integrated DNS resolver within your application, the following steps are required:

  - In 'features.h' insert (or uncomment) #define DNS_SUPPORTED. This will also instruct sevenstaxPPP to request both DNS server addresses (primary and secondary) from the ISP (see below).
  - Because the DNS protocol needs UDP as the transport layer, UDP has to be enabled. This can be done by inserting #define UDP_SUPPORTED in 'features.h'. The local port is set automatically by the sevenstax software.
  - The DNS-server's address. When connecting to an ISP via modem using sevenstaxPPP two IP addresses of different DNS-server, a primary and a secondary one are passed. The primary DNS should preferential be used. The secondary DNS-server should be used only if the primary DNS-server is down.
  - After initialization of sevenstaxTCP call stxTCP_QueryDNS() supplied with the IP address of a DNS and the name of a website.
  - On completion, sevenstaxTCP will notify the application either with NC_DNS_IP or NC_DNS_TIMEOUT.
  - When receiving the event NC_DNS_IP, which means that the DNS has answered, stxTCP_Connect() may be called using this newly gained IP address.

Please Note: Because DNS uses UDP, there is a little restriction on UDP reception during a running DNS query. sevenstaxTCP uses a fixed value as the local port number, which is defined in 'tcpdefs.h' (DNS_LOCAL_PORT) as 11384. If your application listens on this port, data received during an open DNS query is redirected to the DNS resolver and thus never reaches the application. So avoid using the same port number for other UDP traffic.

- *Network Time Protocol (NTP)*

  The Network Time Protocol (NTP) is used to synchronize the time of a computer client or server to another server or reference time source, such as a radio or satellite receiver or modem. It provides accuracies typically within a millisecond on LANs and up to a few tens of milliseconds on WANs relative to Coordinated Universal Time (UTC) via a Global Positioning Service (GPS) receiver, for example. Typical NTP configurations utilize multiple redundant servers and diverse network paths in order to achieve high accuracy and reliability.

  NTP uses UDP on port 123 for both, transmission and reception.

  To access NTP servers with sevenstaxNTP you should proceed the following way:

  1. #define or uncomment NTP_SUPPORTED in 'features.h'
  2. Call stxNTP_Init() once to set-up sevenstaxNTP's internal state.
  3. Call stxNTP_Query() once to start querying the NTP server.
  4. In your main loop or operating system task call stxNTP_Tick() repeatedly in order to keep alive and finish NTP queries.
  5. After a while, the handler procedure receives either an NC_NTP_SUCCESS or NC_NTP_FAILURE event.
  6. On NC_NTP_SUCCESS call stxNTP_Transform() with the 32-bits value in order to convert NTP output into human suitable time values.
  7. On NC_NTP_FAILURE the query can, for example, be repeated with another time server's IP address.

  While in query mode, sevenstaxNTP either post an NC_NTP_SUCCESS or NC_NTP_FAILURE (time-out) message to TCP's handler procedure. On NC_NTP_SUCCESS, the first parameter (lparam) contains the result from the NTP server.

sevenstaxNTP has some user-callable interfaces:

void stxNTP_Init (void)

This function must be called before any other call into sevenstaxNTP. It sets up all internal states. Unlike stxTCP_Init() stxNTP_Init() does not require an own handler function as the results of NTP requests are passed to the TCP handler function. Please see chapter 9 'Application defined Handler Procedure' for details.

*void stxNTP_Query (IPV4 *ntp_server)*

stxNTP_Query() instructs sevenstaxNTP to query a specific NTP server for time. The server's IP address must be submitted as a parameter (a pointer to an IPV4 structure).

void stxNTP_Tick (void)

This function must be called repeatedly to keep sevenstaxNTP alive. Processor time is distributed to sevenstaxNTP using this function.

*void stxNTP_Transform (UINT32 ntp_time, UINT8 *seconds, UINT8 *minutes, UINT8 *hours, UINT16 *year, UINT8 *month, UINT8 *day, UINT8 *day_of_week)*

Using this function one can easily convert an NTP server's answer into its components. The function awaits an NTP response value and seven pointers to variables which receive the results. The NTP response value usually is the first parameter of the handler procedure (lparam) when an NC_NTP_SUCCESS event occurs. Further details can be found in chapter 6 'Public Functions'.

Please Note: While an NTP query is in progress and another call to stxNTP_Query() was made, the first query is canceled. Independently of this, a positive response to the first query results in an NC_NTP_SUCCESS event.

11 Debugging Support

At the source level, sevenstaxTCP contains a simple debugging support through a macro called DPRINT(). Like printf(), DPRINT() has a variable length argument list. To switch on debug outputs you must compile sevenstaxTCP with PROTDEBUG defined in 'features.h' and customise the implementation in 'debug.h' which is platform specific.

Please Note: If you want to use DPRINT() yourself and it follows an if/else statement etc., be sure to enclose it in braces. Not doing so might cause your code to work while debugging but misbehave in release versions.

12 Ethernet and the Address Resolution Protocol (ARP)

This section describes the sevenstaxARP which must be used when sevenstaxTCP runs atop of Ethernet. Readers who don't want to use Ethernet as the link layer can skip this chapter.

Ethernet uses another kind of addressing which is slightly different from those of the Internet protocol. Packets on Ethernet start with so called 'MAC' addresses. These values, six bytes in size, which uniquely identify network interface cards (NICs). The main difference is that MAC addresses are hardware addresses which are (usually but not always) assigned by the NIC's manufacturer, where IP addresses are virtual and can be, in most cases, arbitrary selected by the user.

In order to run sevenstaxTCP on Ethernet the local and remote MAC address must be kept somewhere during the connection. Usually it's no problem to query the local MAC address from the NIC but there is no way to get the remote IP address without an additional protocol.

If sevenstaxTCP runs as a server TCP, the remote MAC address will be extracted from the packet which carries the initial SYN sent by the client (because every packet on Ethernet contains the source and destination MAC addresses). On the other hand, when sevenstaxTCP acts as a client TCP, it must first call the ARP to get the MAC address of the remote station. ARP works the following way:

1. Suppose computer A (which IP address is 192.168.130.2) wants to talk to Computer B (which IP address is 192.168.130.155) using TCP/IP.

2. If computer A already knows the MAC address of computer B (because of a former ARP query) - Go to step 6.
3. Computer A doesn't know the MAC address of computer B. It sends an ARP request to all computers (as an Ethernet broadcast): „Who knows the MAC address of 192.163.130.155?“
4. Computer B receives the request and sends the answer back to computer A (as a directed Ethernet frame): I am 192.163.130.155 and my MAC address is 00-06-f3-72-aa-0d.
5. Computer A stores this Information in a timed cache (associates a timestamp with it) where it lives approximately 20 minutes. This is done to reduce the number of ARP queries because broadcasts are a heavy network load.
6. Computer A now knows the MAC address of computer B. It is now able to build Ethernet frames, put IP packets into it and send them to Computer B.

The sevenstaxARP module exports four public interfaces which must be called by the application.

- *void stxARP_Init (PROTOCOL_NOTIFY_HANDLER h);*

  This function must be called before calling any other function of sevenstaxARP. It sets up the internal state of sevenstaxARP and registers a handler procedure which is used to receive events from sevenstaxARP. The handler procedure can be the same used by sevenstaxTCP.
  Please Note: Before calling stxARP_Init(), the NIC driver should already be up and started.

- *void stxARP_Tick (void);*

  stxARP_Tick() must be called fast and repeatedly to keep sevenstaxARP alive. Because it does the same as stxTCP_Tick() does for sevenstaxTCP, this function can be called from the same location of your app, where stxTCP_Tick() is called.

- *void stxARP_Query (IPV4 ip);*

  A call to this function instructs sevenstaxARP to query the MAC address of the corresponding IP address. sevenstaxARP will send some ARP query packets over the network until the remote ARP answers. If no answer was detected during a short time (default is 2 seconds), sevenstaxARP will give up and notify the application of an unsuccessful attempt.

- *BOOL stxARP_ProcessPacket (PACKET_DESC *pkt);*

  This function must be called from the NIC driver to deliver ARP packets to sevenstaxARP. Because sevenstaxARP does not look at the type of packet, the NIC driver must call stxARP_ProcessPacket() only when it has recognised the type of packet (which must be 0x0806 for ARP). stxARP_ProcessPacket() returns TRUE either if the packet was a valid ARP query or it was an answer to a query sevenstaxARP has sent before. Otherwise, stxARP_ProcessPacket() returns FALSE either if the the IP address of the packet doesn't match the local one or the ARP opcode is an unknown value.

In addition, sevenstaxARP needs to access two variables for both, the local and remote MAC addresses. These variables have connectors in 'tcpconnector.h' and are called REMOTE_MAC_ADDRESS and LOCAL_MAC_ADDRESS. Usually, they reside in the NIC driver, but that's not a requirement. The local mac address must be valid before sevenstaxARP starts its work, this implies that the NIC driver already runs (see above). Both variables are six bytes in length and are stored in network byte order. After a successful call to stxARP_Query(), sevenstaxARP stores the result in REMOTE_MAC_ADDRESS.

At last, sevenstaxARP must know the local IP address in order to build appropriate ARP packets. This is an IPV4 structure which is located in the sevenstaxTCP module (see chapter 8, Local IP address).

The following example shows how to incorporate ARP into your application:

```
// Flag to detect ARP response
BOOL bARP_response = FALSE;

// This function receives all events from ARP and PPP
void handlerfunc (NOTIFY_CODE a, UIN32 b, UINT32 c)
{
     case NC_ARP_HOSTMESSAGE:
     if (b == ARP_OP_REPLY)
     {
            // Got ARP response
            // REMOTE_ETHERNET_ADDRESS is valid now
            bARP_response = TRUE;
     }
     break;

     case NC_ARP_TIMEOUT:
     // We got no response, TCP cannot connect
     // Do something to handle this error
     break;

     // Handle TCP related messages
     case NC_TCP_...
     .....
     break;
     ......
}

// Here we go....
void main (void)
{
     IPV4 ip;                              // The IP address we want to connect to
     ip.ipb.b1 = 192;                      // This should be 192.168.130.68
     ip.ipb.b2 = 168;                      //      ''
     ip.ipb.b3 = 130;                      //      ''
     ip.ipb.b4 = 68;                       //      ''

     // Set up our own IP address to 192.168.130.155
     LOCAL_IP_ADDRESS.ipb.b1 = 192;
     LOCAL_IP_ADDRESS.ipb.b2 = 168;
     LOCAL_IP_ADDRESS.ipb.b3 = 130;
     LOCAL_IP_ADDRESS.ipb.b4 = 155;

     stxETH_Init();                // Start the NIC
     stxARP_Init (handlerfunc);    // Init ARP and register handler proc
     stxTCP_Init(handlerfunc);     // Init TCP, using the same handler
                    // procedure
     stxARP_Query (ip);            // Instruct ARP to ask for the
                                   // remote MAC
     while (1)
     {
            stxETH_Tick();                // Do NIC related tasks
            stxARP_Tick();                // Keep ARP alive
            stxCP_Tick();                 // Keep TCP alive
            // Wait for an ARP response ....
            if (bARP_response == TRUE)
            {
                   // We're able to connect  - now do it
                   bARP_response = FALSE;
                   TCP_Connect (ip, 80);
            }
            // Call routines elsewhere in your application
            ......
     }
}
```

13 Restrictions

sevenstaxTCP is focused on systems with very small resources, to give them relevant TCP/IP functionality. Therefore – in comparison with a full featured stack implementation of a PC software f.e. - it has some restrictions in usage and capabilities, the user should know:

#	*Restriction*	*Details*
1	Not aware of interrupts	If parts of sevenstax embedded protocols are called from within an ISR, synchronization must be done by the user.
2	Not full RFC compliant	To give maximum capabilities within small system resources, sevenstax embedded protocols fulfil only basic and most relevant parts of the RFC specifications.
3	Multiple TCP connections atop Ethernet	The ARP implementation currently has no cache. sevenstaxARP stores only one MAC/IP address pair. sevenstaxTCP issues an ARP query for every IP address which it doesn't know. This works with multiple TCP connections, but is inefficient. It will be point of improvement in next release.
4	Transmission Speed	Because sevenstaxTCP has no buffering nor buffer management, every TCP packet must be ACK'd by the receiver before the next one is send.
5	Fixed window size	Outgoing TCP packets always have a fixed window size, which can be configured at compile time (TCP_WINDOW_SIZE in 'tcpdefs.h').
6	No buffering of any kind	Because sevenstaxTCP has no data buffers, when sending data over a TCP connection using stxTCP_Write() or stxTCP_WriteBulk(), the content must exist and must not be changed, until the application receives a feedback (NC_TCP_TXREADY). Therefore, data to send should be in static memory.
7	IP fragmentation	sevenstaxTCP does not support IP fragmentation. This is no disadvantage for TCP/IP, but for UDP packets: Only the first part of fragmented UDP packets are received and used. sevenstaxTCP also never generates fragmented IP packets by itself.
8	Replies from name servers (DNS)	Because DNS packets are send over UDP and sevenstaxTCP throws away UDP fragments, DNS only works if either the reply (from the name server) is not fragmented or the wanted IP address is in the first fragment. For very large sites, such as www.google.com, name server replies doesn't fit into a single UDP packet. But In the majority of cases, an IP address is included and found in the first fragment.
9	IP address from name servers (DNS)	sevenstaxTCP only uses the first IP address found in a name server's reply.
10	ICMP	The only part of ICMP, which is currently implemented into sevenstaxTCP, are ICMP echo replies (PING).
11	Use of Ethernet and PPP	The current version of sevenstaxTCP does not support use of PPP mode and Ethernet mode at the same time, because the #define RUNS_ON_ETHERNET is responsible for mutually exclusion. It will be point of improvement in next release.

sevenstaxTCP is object of permanent improvement. Please contact info@sevenstax.de to get more information about new TCP/IP features given in next release version.

14 Description of Files

File	Description	sevenstax TCP	sevenstax optional
tcp.c	sevenstaxTCP source code	x	
tcp.h	sevenstaxTCP's public functions prototypes	x	
tcpdefs.h	Constants and definitions for sevenstaxTCP	x	
tcpconnector.h	sevenstaxTCP lower API function prototypes to be used with Ethernet or PPP interface	x	
pktdesc.h	Packet analyse types and macros	x	
standard_types.h	Target system basic types	x	
ipv4.h	IP address structure and supporting macros	x	
wordaccess.h	Target systems word/byte alignement	x	
features.h	Enable/disable sevenstaxTCP features at compile time	x	
notifycodes.h	Events and definitions regarding to handler procedures for user API	x	
runtimelib.h	Connectors to c-library functions.	x	
runtimelib.c	Replacements for c-library functions.		x
debug.h	Debug functions prototypes		x
arp.c	Implements the Address Resolution Protocol (ARP)		x
arp.h	Prototypes of ARP public functions		x
ntp.c	Network Time Protocol source code		x
ntp.h	Network Time Protocol definitions and prototypes		x
ethernet.h	Ethernet driver code, header file if used for ethernet		x
ethernet.c	Ethernet driver demonstration code, simulation under Windows NT / 2000, uses packet driver (not included)		(x)
main_eth.c	TCP main() demo code, sending HMTL string when TCP connected		(x)

(x) These files are only needed in the test environment

15 Change History

Vers.	*Date*	*by*	*Change description*
2.88	21/08/2002	pe	Base version
2.91	12/12/2002	pe	Improved API
3.01	16/01/2003	pe	Additional features
3.02	13/02/2003	krz	Reviewed
3.03	21/02/2003	krz	some corrections
3.11	26/02/2003	pe	restrictions added

Copyright (c) 2003 by sevenstax GmbH
This document is an intellectual property of sevenstax GmbH. Unauthorized copying and distribution is prohibited.